A Generic Computer Support for Concurrent Design

Jacques Lonchamp
LORIA, BP 254, 54500 Vandœuvre-lès-Nancy, France
Jacques.Lonchamp@loria.fr

Abstract

Concurrent Design (CD) involves collaboration, coordination, and information-based co-decision making within a potentially distributed multifunctional team. This paper shows that a generic process-centered environment kernel, based on fine grain and decision-oriented task modeling, using customizable product models, providing capabilities for task model refinement at run time, and true collaboration support, is a good candidate for building dedicated computer aided CD environments. DOTS ('Decision-Oriented Task Support'), a Java prototype of such a generic environment kernel, is described in this paper and its usage in the CD application domain is discussed.

1 Introduction

The research described in this paper deals with computer support for *Concurrent Design* (CD), i.e., for the early phases of the Concurrent Engineering process. During CD, multifunctional teams, possibly distributed in time and space, work together for designing some product. In such a setting an efficient support for *collaboration, coordination, and information-based co-decision making* is needed.

Collaboration can be defined as a group process in which the group has common goals and produces one unanimous result (i.e., contributions are no longer attributes to group members, and the whole group takes responsibility for the result). Information sharing is the basic prerequisite for collaborative work: it implies common data models, shared data, and controlled access to them [1]. But data sharing is not sufficient for establishing and maintaining a true 'shared understanding' among the participants. Knowledge integration is also required, for instance through collective idea generation and discussion. In fact, a 'common information space' is negotiated by the actors involved [2].

Coordination is concerned both with the synchronization of activities (sometimes called 'activity-level coordination' [3]) and the synchronization of concurrent access to shared objects (called 'object-level coordination' [3]). Design processes are complex and intellectually demanding, and cannot be completely captured in a fixed process definition beforehand. To achieve flexible activity-level coordination, facilities are needed to support design process modeling, model execution, and (possibly collaborative) model refinement at run-time.

Co-decision making is central to CD [1]. It implies first allocation and sharing of responsibilities among the participants, and secondly, flexible support for various co-decision making processes.

The objective of this paper is to show that a generic process-centered kernel, based on *fine grain and decision-oriented task modeling*, using customizable product models, providing capabilities for task model refinement at run time, and true collaboration support, is a good candidate for building dedicated computer aided CD environments, because it takes in account explicitly the three aspects above-mentioned. The paper describes DOTS ('Decision-Oriented Task Support'), a Java prototype of such a generic kernel, and discusses its usage in the CD application domain.

We begin the paper by defining the general objectives of DOTS project. We then describe, in section 3, its conceptual meta model, with a particular emphasis on the argumentative reasoning aspect. In section 4, we discuss the architecture and usage of the current prototype, and we offer a small example of use in the CD domain. The paper closes by outlining further research directions.

2 The Objectives

This section summarizes the main objectives and requirements of DOTS project. They were elaborated with different application domains in mind.

(1) The system should support a small group of people (from two to less than ten) participating in a collaborative task, mainly distributed in time (asynchronous) and space; 'occasionally synchronous' work should also be considered. The coordination with other individual or collaborative tasks, i.e., the classical workflow aspect, is left outside DOTS prototyping effort because two other projects in the same research team focus on different aspects of Java-based workflow support [4, 5].

(2) The system should support a range of tasks, through a generic infrastructure, parameterized by a task model; this model should be often the customization of a basic library model, with some aspects that could remain unresolved until the execution (dynamic task model refinement).

(3) The system should provide an efficient assistance in three domains: guidance (i.e., task performance assistance and task model refinement assistance), argumentation and decision assistance, and group awareness (both asynchronous and synchronous).

(4) The project should provide a complete system for initial model development (editing, compiling, and verifying), system deployment support (installing and instantiating), and model execution and dynamical refinement. In simple cases, it should be possible to generate a fully operational customized system from the task model and from the standard kernel. In more complex cases, the environment designer should have to customize the generic product and tool types, and rarely should have to work at the generated code level.

(5) Both the entire infrastructure (client, server, development tools) and the generated code should be Java code, mainly for taking advantage of Java platform portability property.

(6) The project should provide a library of generic task models for brainstorming, collective review/ inspection, collective confrontation/merging of conceptual descriptions, free argumentation (i.e., emulation of an argumentation groupware), etc.

3 The conceptual meta model

In DOTS, a task model is described according to four perspectives: activity-oriented, decision-oriented, product-oriented, and organization-oriented, as shown in Fig. 1. Of course these perspectives are not independent and there are many relationships between them.

The next four subsections describe these perspectives and the last subsection emphasizes the argumentative reasoning aspect.

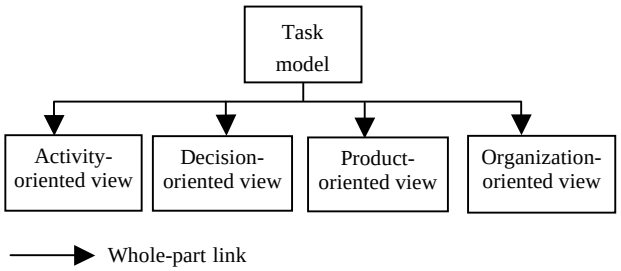


Figure 1 The overall conceptual organization of a task model.

3.1 The activity-oriented view

A collaborative task is structured into phases, in which the nature of the work (see section 3.2), the number and the identity of the participants can vary. During a phase (individual or collective) decisions are taken that modify the products under construction: operations are the elementary chunks of work, triggered by a decision. During a phase, participants can also freely access tools for performing activities not related to decisions (e.g. through query tools, server-side scripts, client-side external tools). The activity-oriented view of the task model mainly describes the phase types, the operation types, and the tool types (see Fig. 2).

When the task model is instantiated, a graph of phase instances is built, with phase precedence links. This instantiation can take place either statically (i.e., before execution) or dynamically (i.e., during execution).

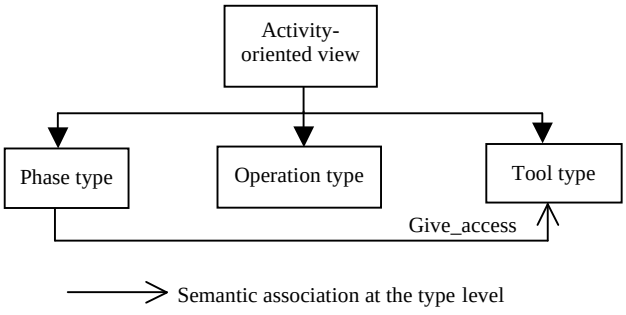
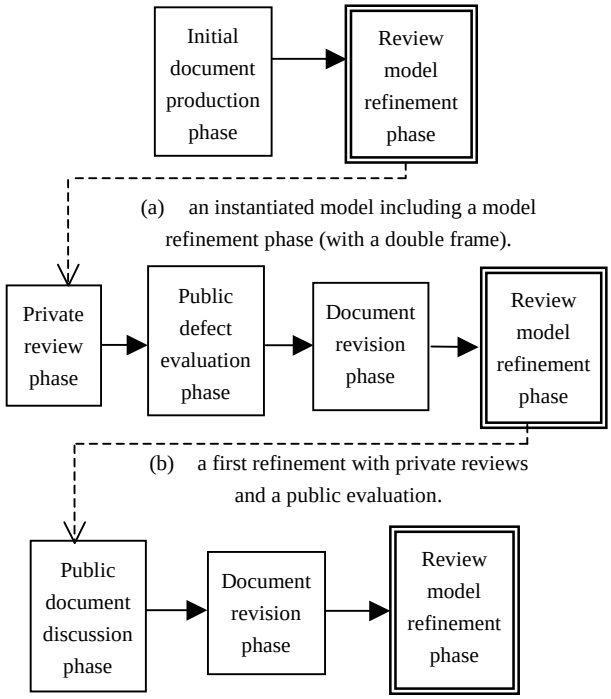


Figure 2 Main elements of the activity-oriented view.

Fig. 3(a) shows such an instantiated model describing how a design document is reviewed. First, the initial document is written down (here, we do not describe this

task in details). Then, during the 'Review model refinement phase', a review model is chosen (individually or collectively) and dynamically instantiated. In the first refinement solution of Fig. 3(b), defects are first proposed individually and privately; then, during the 'Public defect evaluation phase', the proposed defects are collectively discussed and evaluated, i.e., accepted or rejected; finally, the document editor modifies the document in accordance with the review results. Then a new review can take place, whose model is one more time dynamically chosen and instantiated. A simpler review, without private phases can be sufficient at this stage as depicted by Fig 3(c).



(c) a second refinement with a public discussion.

Figure 3 A task model instance and two possible refinements.

3.2. The decision-oriented view

An issue is a problem that must be solved, generally concerning the products under construction. But the choice between different task refinements, as discussed in the previous section, is also an issue.

In most tasks, the different issue types are progressively taken into consideration. A phase is mainly defined by the subset of the task issue types taken in consideration at this stage.

Several option types specify how the issue type can be solved (e.g., AcceptDefectOption, and RejectDefect Option for Evaluate DefectIssue).

At the level of the task execution, i.e., at the level of the instances, users argue about the options of each issue instance. The decision takes (more or less) into account this argumentation in relation with the resolution mode of the issue type (see below).

Arguments are instances of a single Argument type and include a free textual rationale. Participants argue about the options and about the arguments themselves. They can also give qualitative preference constraints between the arguments (MoreImportantThan or >, LessImportantThan or <, EquallyImportantThan or =). Participants can also argue about the constraints: constraints as arguments are refutable. All the time, the system computes 'the best solution' in accordance with the current argumentation state (see section 3.5); but the actual decision is generally kept independent from the argumentation.

To each option type can be associated an operation type. The operation is triggered when the option is chosen. This operation can modify:

- a product or one of its components (e.g. add a defect to the list of proposed defects); this product evolution can in turn suggest new issues;
- the task content (e.g. dynamical creation of issues, options, phase instances, tools).

An option can also terminate a phase (see Fig. 4).

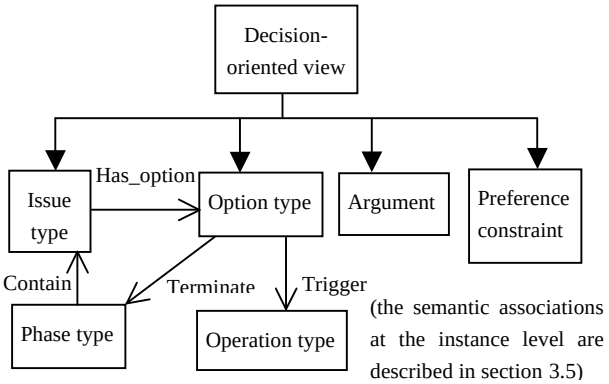


Figure 4 Main elements of the decision-oriented view.

The main characteristic of an issue type is its resolution mode:

- individual: the issue is solved by its creator;
- individualPrivate: similar to the previous mode but the issue and its consequences are only visible by the creator of the issue;

- collectiveDemocratic: the resolution is collective because the solution is necessarily 'the best solution' proposed by the system (see section 3.5) and at least two different users must take part in the argumentation;
- collectiveAutocraticWithoutJustification: the argumentation is collective but the choice is individual (autocratic); the choice is independent from the best solution proposed by the system and does not require any formal justification;
- collectiveAutocraticWithJustification: the choice is autocratic but requires a formal justification: an explanation step follows the argumentation in which only the decision-maker can argue in order to make the best solution equal to his/her own solution.

From a dynamical point of view, the life cycle of an issue is a sequence of interactions (expressed below as regular expressions):

- RaiseIssue: creates the issue instance (generally with parameters) and the corresponding option instances,
- (GiveArg | GiveConstr) $^{+}$: creates the argumentation tree,
- SolveIssue: solves the issue and triggers the operation associated to the chosen option (this operation generally makes use of the issue parameters).

There exist two important simplified cases. For an individual issue with a single option, only RaiseIssue is necessary (all the remaining is automatic): the issue is just an elementary action. For an individual issue with several options, only RaiseIssue and GiveArg are necessary: the issue is just an individual choice between several elementary actions, the argument can be understood as the rationale for the individual choice.

3.3. The product-oriented view

A product includes components at different levels of granularity. Products are currently specialized into textual product, list product, image product, and graph product (see Fig. 5). A parallel classification exists for tools (e.g. textual viewers, list viewers, image viewers, graph viewers). A minimum set of features is provided by these generic types (such as automatic graph layout methods); more specific features can be introduced through specialization or choice among predefined constraint verification rules, in the spirit of generic concept map editors such as [6]. And/or goal structures or design rationale descriptions are examples of specialized graphs useful in CD task models, which can be provided by a customized kernel. Documents and tools can be instantiated either statically or dynamically.

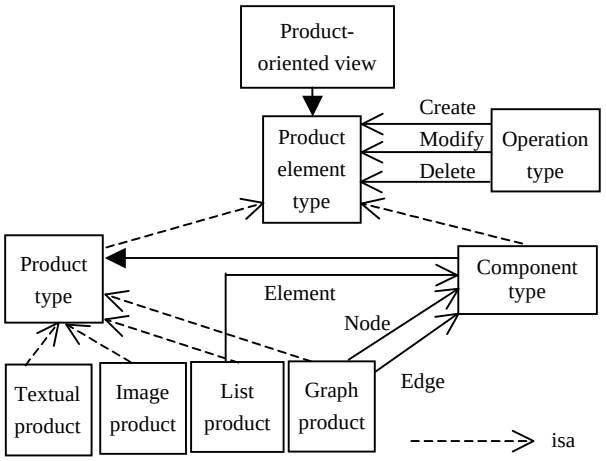


Figure 5 Main elements of the product-oriented view.

3.4. The organization-oriented view

Actors (currently restricted to human participants) play roles. Role types define what actors are allowed to do (see Fig. 6). Actors are instantiated statically or dynamically

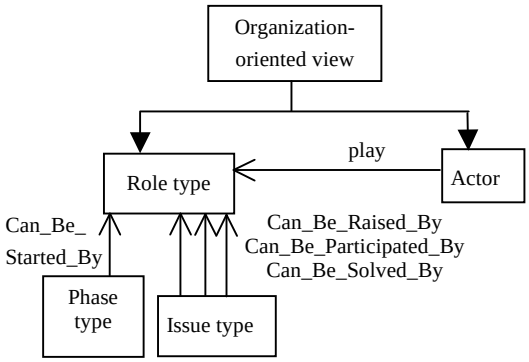


Figure 6 Main elements of the organization-oriented view.

3.5. The argumentative reasoning aspect

The system provides participants means of expressing their individual arguments and qualitative preferences, the aim being the selection of a certain solution. We discuss the evaluation procedure in two steps related to the absence (presence) of qualitative preference constraints.

Without preference constraints. The issue, the options, the arguments 'for' and 'against' the options, the arguments 'for' and 'against' the arguments form an argumentation tree. A score and a status (active, inactive) that derive from the score characterize each node of the tree. The score of a father node is the sum of the weights of its active child nodes that are 'for' their father minus the sum of the weights of its active child nodes that are 'against' their father. If the score is positive the node is active otherwise it is inactive. Only status propagates in the tree (because scores have no global meaning).

Without preference constraints, all nodes have the same weight (for instance 5, middle of the arbitrary interval 0-10 used in the next subsection, where 10 denotes the maximum importance). Leaves are always active. The preferred option (best solution of the issue - one or several) has the maximum score among all the options (see Fig. 7).

With preference constraints. Preference constraints are qualitative preferences between arguments of different options (global constraint) or between arguments of a same father argument (local constraint). One argument (source) is compared to the other (destination).

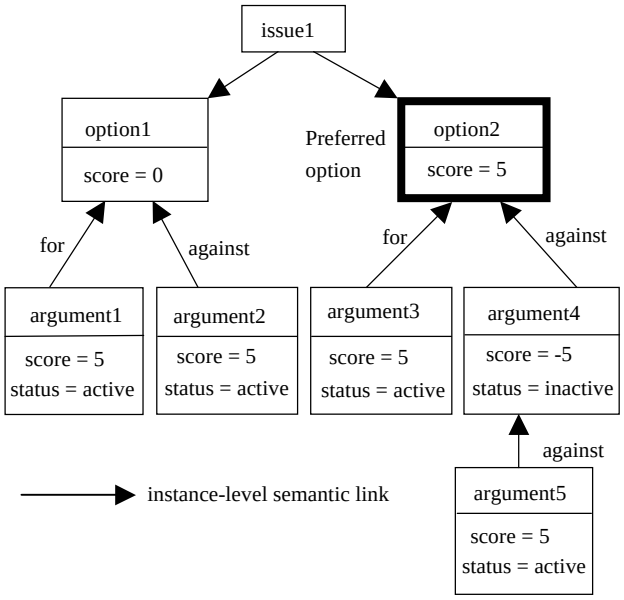


Figure 7 An argumentation tree without constraint.

Consistency is evaluated when the constraint is created and evaluated when another constraint becomes inactive.

To each constraint is associated a ConstraintIssue with three positions: MoreImportantThan (>), LessImportant Than (<), EquallyImportantThan (=). A constraint is active if both the arguments are active, if one of its options is chosen (score strictly higher than the others) and if it is consistent with the other constraints (the evaluation is based on a path consistency algorithm). For instance, if arg1, arg2, and arg3 have the same father, and arg1 > arg2, arg2 > arg3, then arg3 > arg1 is inconsistent. This last constraint becomes (provisionally) inactive.

The weight of all arguments having the same issue as grand father (global constraint) or the same argument as father (local constraint) is computed by the following heuristics:

- all > relationships are defined by propagating them along the = relationships,
- for each argument arg involved in a constraint:
  . its max weight is computed, by subtracting 1 (starting from 10) for each argi such argi > arg;
  . its min weight is computed, by adding 1 (starting from 0) for each argument argj such arg > argj;
  . its final weight is computed as the average of its max and min weights.
  . the weight of an argument not involved in any constraint is kept to the average value (5);
- the rules of the previous item for computing the scores and the status are applied with these computed weights.

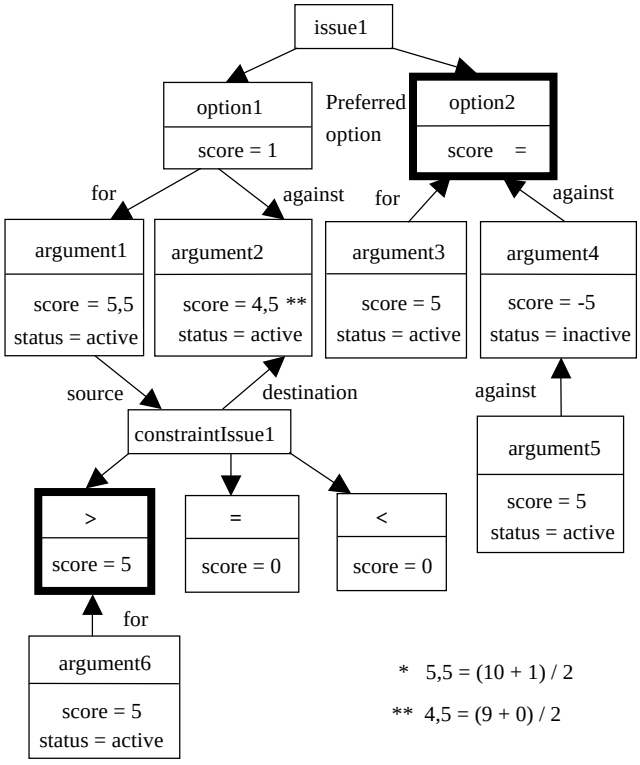


Figure 8 The argumentation tree of Figure 7 after introducing a constraint.

After each modification the whole tree is re-evaluated: for instance, inactivating an argument can re-activate a constraint that was inactive because it was inconsistent with the former constraint, which changes the status of an argument, which propagates on the upper level, and so on. As an illustration, in Fig. 8 a constraint is added to the argumentation tree depicted by Fig. 7. This argumentative reasoning technique is based on both Hermes and Zeno approaches [7,8].

4. DOTS PROTOTYPE

4.1.The system architecture

The system has a client/server architecture around an object database with a Java API. The database provides persistency, consistency, safety, and security. Communication and notification aspects are managed by a specific Java infrastructure. Persistency is conforming to the ODMG Java binding: persistent classes are declared statically and pre-processed before the Java compiler is called. This makes impossible dynamical schema evolutions. So, in the current prototype, we have chosen to generate (transparently) one separate database for each version of a task model. All these databases are accessed through a 'super base', and can be located on different machines. Each database contains the kernel, one task model version, and all the task instances conforming to this model. If a task model is changed, it is possible to run instances of these two different versions located in the two different databases.

The client is independent of the task model. It is written in Java and swing. The development and deployment environment includes three other tools, all written in Java and swing: a development tool (editor and compiler), an instantiation tool, and a static analyzer of instantiated models.

4.2. The main functionalities

The user enters the system with a registered user name (created with the instantiation tool and kept in the 'super base'), in one of the task instances in which he/she plays a role. The user can then act in accordance with the task model, the current task status, and his/her role. The user receives a threefold assistance: guidance (how to perform the task and how to refine the task model), argumentation and decision assistance, synchronous and asynchronous group awareness.

For task execution, the user can obtain the list of possible next interactions in accordance with the current task status and his/her role: issue types that can be raised, issue instances that can be participated in, and solved, phase instances that can be started, etc. Obviously only those possible interactions are accepted by the client. The user can also access to different textual and graphical views of the task model and of the task history (with colors highlighting for instance the active elements). Refining a task model is solving an issue that defines the different available solutions. As for each issue some static guidance is provided. Dedicated tools (e.g. query tools) can also provide dynamical information to make the choice easier.

At the argumentation level, the best option of each open issue is shown in color in the graphical view, as the active arguments and constraints; scores and weights can be displayed. The user can also list all open issues that are currently inconclusive (no option with a higher score than the others).

The main mechanism for asynchronous awareness shows what has evolved since the last connection of the same user in the same task (textual list, and specific color in all graphical views). For "occasionally synchronous" work, the user can obtain the list of all the connected users in the same task, can receive the notification of all constructive public actions from these other users in a notification window, and is warned when a document or a graphical representation becomes out of date (its background color changes).

Fig. 9 shows a client during the evaluation phase of a simple review, whose model is shown in window 1. Window 2 is the log window that contains the results of the interactions (here, a "what can I do? " request). Window 3 is the notification window: one can see that another user has logged in and has proposed a new defect. Window 4 is the NotYetEvaluatedDefect viewer tool and its content has become out of date after the creation of the new defect (the out of date marker is the dark background color). Window 5 shows graphically the current state of an issue; this description as the task description in window 1 is up to date (white background). Icons with a colored frame highlight active phases in window 1 and active nodes in window 5.

4.3. A CD example

We consider a multifunctional team of domain experts participating in a collaborative goal-directed acquisition task. The objective is to build collectively a goal-subgoal structure for a particular system (and/or graph).

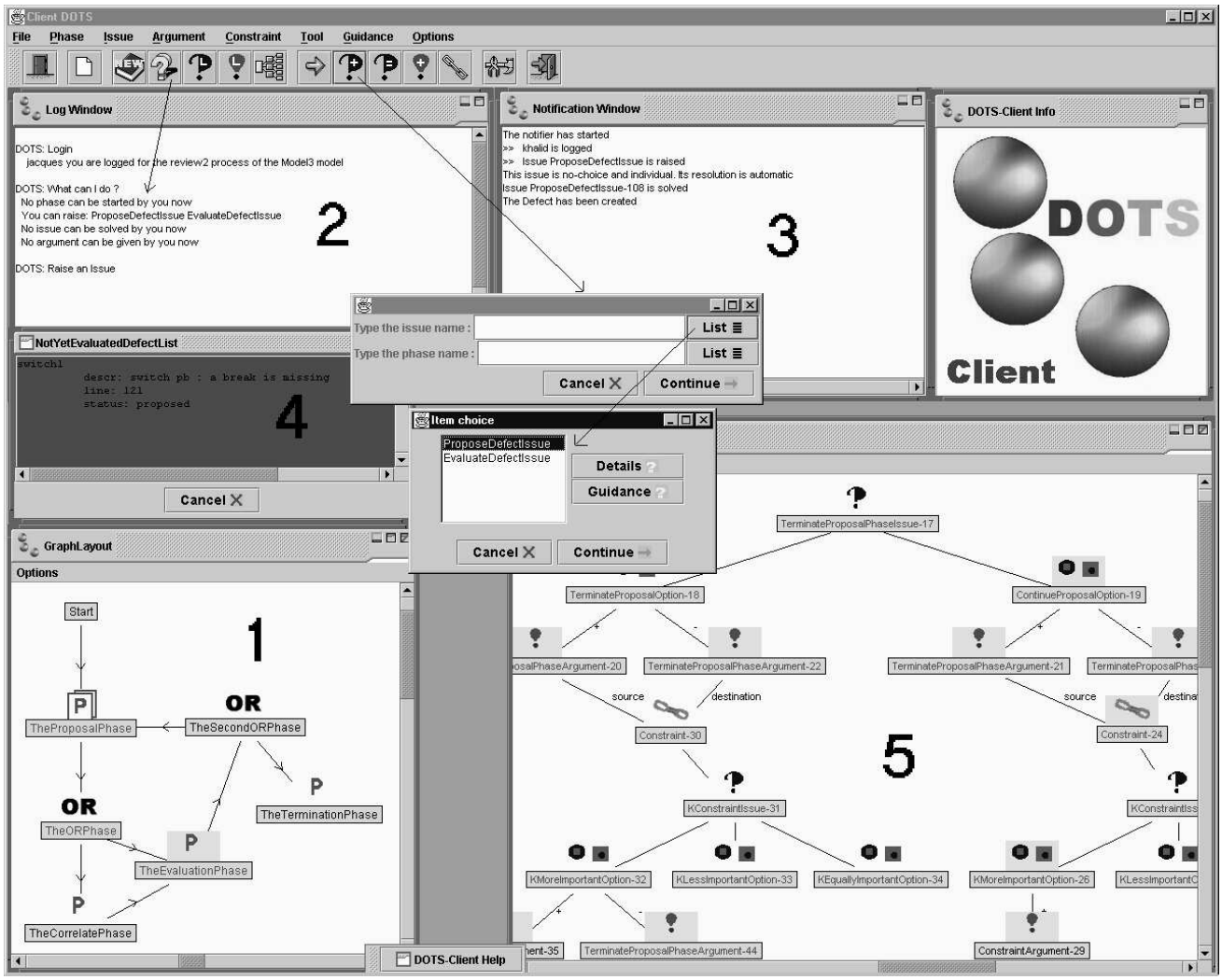


Figure 9 DOTS Client.

The task model organization. Classical strategies can apply: for instance, a private brainstorming phase for eliciting a maximum number of goals, followed by a public discussion phase for searching duplicate or irrelevant proposals, followed by an initial goal structure construction phase (for instance by the team leader), and terminated by an iterative review-revise cycle for improving the initial proposal.

First, DOTS provides the ability to manipulate graphical goal structures (graph management, graph layout, node expansion, etc.): the generic concept map library can be customized for this kind of graph, with customized presentation characteristics, and basic or specific properties verification.

Private brainstorming can be supported. In this mode, each participant cannot see the proposals of the others. Relaxed privacy is also possible (e.g. with a tool allowing to see a random choice of proposed goals), as public brainstorming. Proposing a new defect is just solving individually a ProposeGoalIssue, the argument being the rationale of the proposal.

In the public discussion phase, participants can raise issues for resolving duplicates, and for challenging irrelevant proposals. These issues are discussed, possibly with a very complex argumentation tree, and solved in accordance with some resolution mode (section III-B). Asynchronous and 'quasi synchronous' (i.e. through immediate notifications and out of date markers) working modes are available.

The initial construction and the iterative review-revise cycle are similar to the process discussed in section III-A.

By introducing in the goal structure notation objects which are processor for actions and the 'is responsible for' and 'wishes' relations, more assistance can be provided through goal reduction heuristics [9]. In DOTS, dedicated query tools can suggest possible goal reductions (such as the list all goals for which the responsibility is shared among several agents that are candidate for further reduction).

The task model specification. A task model is divided in two parts which describe the model specific entity types (specialization of phase, issue, option, role, document, tool, component, operation types) and the relationship types between them (Contain, Give_Access, Has_Option, Trigger, Terminate, Can_Be_Started_By, Can_Be_Raised, Can_Be_Participated_By, Can_Be_Solved_By, Create, Modify, Delete – see section 3).

The core part of each task model specifies issue types and related operation types that aim at changing product components. An issue type has a name, a resolution mode (KIND), a boolean saying if the issue is a simple alternative (in this case all the argumentation takes place on a single option, otherwise it is necessary to argue 'for' and 'against' all the options), two booleans saying if only a single instance or a single active instance can exist, the parameters of the issue (with the interaction messages and possibly OQL queries for generating list boxes), a textual description, and optionaly a static guidance on how to choose among the different option types:

```
<issue-type-name>
  KIND <mode>
  TRUE_ALTERNATIVE <boolean>
  UNIQUE_INSTANCE <boolean>
  UNIQUE_ACTIVE_INSTANCE <boolean>
  PARAMETERS
   (LABEL<message>
      [QUERY <OQL-query>]";" )*
  END_PARAMETERS
  DESCRIPTION <text>
  [GUIDANCE <text>]
```

The document types and the tool types are classified by content: text, image, list (giving the component type name), and graph (giving the node type name, and the edge type name). The component types can have attributes:

```
ATTRIBUTE_TYPES
  (<type-name> <attribute-name>
    ["=" <constant>] ";" )*
END_ATTRIBUTE_TYPES
```

The operation types are described through a list of elementary action specifications (of type CREATE, MODIFY, DELETE, EXECUTE - a script -, MAILTO):

```
ACTION
  (<action-specification> ";" )*
END_ACTION
```

For instance, the ChallengeGoalIssue type of the public discussion phase is specified by:

```
ChallengeGoalIssue
 KIND collectiveDemocratic
 TRUE_ALTERNATIVE true
 UNIQUE_INSTANCE false
 UNIQUE_ACTIVE_INSTANCE false
 PARAMETERS
   LABEL "Give the goal identifier: "
     QUERY SELECT * FROM Model.GoalExtent
       WHERE status = "proposed";
   // parameter 1 with a list box
 END_PARAMETERS
 DESCRIPTION "Collective evaluation of one
    of the individually proposed goal"
 GUIDANCE "Choose either to reject the
    goal or to keep it"
```

Two options types are associated to ChallengeGoalIssue: KeepGoalOption, and RejectGoalOption. RejectGoal Option can trigger an operation of type Invalidate GoalOperation:

```
InvalidateGoalOperation
  ACTION
    MODIFY Goal PARAM(1) WITH status =
      "refused";
    // PARAM(1) is a reference to the
    // parameter 1 of the associated issue
    // type
  END_ACTION
```

In more complex cases, the component can also be retreived through an OQL query, possibly including references to the issue parameters. Moreover, several actions in the same operation can be related through local variables. The second example below shows the dynamical creation of a goal substructure, of the graph document associated to this component, and of its viewer tool:

```
CreateGoalSubstructure
  ACTION
    CREATE GoalStructure AS gstruct
      // local variable gsstruct
      WITH IName = PARAM(2)
      WITH enclosingGoalStructure =
```

```
      PARAM(1);
    CREATE GoalSubGraph AS gsgraph
      // local variable gsgraph
      WITH referent = gsstruct
      WITH referentAttribute = "enclosing"
      WITH componentIconPath =
         "Images/GoalGraph.GIF";
    CREATE GoalGraphViewer
      WITH TheGraph = gsgraph
      WITH IName = PARAM(3);
END_ACTION
```

The local variables gsstruct and gsgraph are useful for linking the three dynamically created components.

5. CONCLUSION

The generic infrastructure described in this paper aims at assisting participants of decision-oriented collaborative tasks. The approach is mainly based on fine-grain modeling of these tasks and the use of different assistance techniques: guidance, argumentative reasoning, group awareness.

To sum up, DOTS makes a synthesis of classical features of flexible generic process-centered environments, of argumentation and decision support systems, and of synchronous/asynchronous groupware systems.

Most of existing CD environments just act as a repository for and a controller to design artifacts (e.g. CASCADE [10], CoConut [11], Flecse [12], SHARE [13]). These systems do not provide activity-level coordination support (process support). On the opposite, Workflow Management Systems (WFMS) support predefined procedures and sometimes also ad-hoc processes, but do not have adequate support for synchronous or asynchronous collaborative and co-decision making activities. Therefore, some approaches aim at integrating more or less tightly WFMS and collaboration tools (such as WoTel [14] or iDCSS [15] in the concurrent engineering domain). Only few systems truly integrate collaboration and coordination facilities. SCOPE [16] is the closest system from DOTS: it provides flexible support for specification, modification, monitoring, and execution of session-based collaborative processes. However, DOTS argumentation support has no counterpart in SCOPE.

From the concrete feasibility point of view, a previous mock-up system written in Smalltalk had already convinced us of the approach interest, in particular through a real size experiment [17]. The fundamental ‘issue-argument-decision-operation’ cycle seems easy to understand and use, even for inexperienced end users.

Our central claim is that building dedicated computer aided environments (in the CD application domain for instance) is made easier with DOTS. The main part of the work is to write the task model, possibly with model refinement alternatives. It is worth noting that most models will be constructed as a combination and customization of generic building blocks. Another part of the work is to tailor the generic product and tool types, such as OQL-based query tools for dynamic guidance. What is given for free are the client/server architecture, the client interface, the secure server storage, the process engine, the argumentation engine, the guidance and awareness capabilities.

In the next future, we plan to use DOTS for studying in depth and systematically several collaborative tasks that constitute the basic building blocks of many cooperative processes, such as concept graph co-authoring, and concept graph merging.

In a longer perspective we want to investigate other kinds of assistance, that could be plugged in DOTS kernel. For instance, the collaboration could take place not only between human participants, but could be assisted by software agents customized for participating to *issue instances production* and, possibly, to *issue instances evaluation and resolution*.

Acknowledgements

We would like to thank all members of the ECOO INRIA project for helpful discussions.

References

[1] R. Reddy, K. Srinivas, V. Jagannathan, R. Karinthi, "Computer Support for Concurrent Engineering", IEEE Computer, Vol 27, pp 12-16, 1993.

[2] K. Schmidt, L. Bannon, "Taking CSCW Seriously ", CSCW Int. Journal, vol 1, 1/2, Kluwer Academic Publisher, pp 7-40, 1992.

[3] C. Ellis, J. Wainer, "A Conceptual model of Groupware", in Proceedings of ACM CSCW'94 , pp. 79-88, 1994.

[4] K. Benali, M. Munier, C. Godart, "Cooperation models in co-design", International Journal of Agile Manufacturing (IJAM), 2, 2, 1999.

[5] G. Canals, P. Molli, C. Godart, "Tuamotu: support for telecooperative engineering applications with replicated versions", IGROUP Workshop, WorkingPaper B-56, Oulu

University Press, www.idi.ntru.no/~igroup/proceedings/canals.doc, 1998.

[6] R. Kremer, "Constraint Graphs: a concept map meta language", PhD Thesis, University Of Calgary, www.cpsc.ucalgary.ca/~kremer/dissertation/index.html, 1997.

[7] N. Karacapilidis, D. Papadias, "A group decision and negotiation support system for argumentation based reasoning", in Learning and reasoning with complex representations, LNAI 1266, Springer-Verlag, 1997.

[8] N. Karacapilidis, D. Papadias, T. Gordon, "An argumentation based framework for defeasible and qualitative reasoning", in Advances in Artificial Intelligence, LNAI 1159, Springer Verlag, pp 1-10, 1996.

[9] A. Dardenne, S. Fickas, A. van Lamsweerde, "Goal-directed concept acquisition in requirements elicitation", in Proceedings of 6th Int. Workshop on Software Specification and Design (IWSSD), pp 14-21, 1991.

[10] C. Branki, "The acts of Cooperative Design", CERAs, 3, 3, pp 237-245, 1995.

[11] U. Jasnoch, H. Kress, K. Schroeder, M. Ungerer, "CoConut: Computer-Support for Concurrent Design using STEP", in Proceedings WetIce, 1994.

[12] P. Dewan, J. Riedl, "Toward Computer-Supported Concurrent Software Engineering", IEEE Computer, Vol 27, pp 17-27, 1993.

[13] G. Toye, M. Cutkosky, L. Leifer, J. Tenenbaum, J. Glicksman, "SHARE: A Methodology and Environment for Collaborative Product Development", in Proceedings of IEEE Infrastructure for Collaborative Enterprises, 1993.

[14] M. Weber, G. Partsch, S. Hoeck, G. Schneider, A. Scheller-Houy, J.Schweitzer, "Integrating Synchronous Multimedia Collaboration into Workflow Management", in Proceedings of GROUP'97 , pp 281-290, 1997.

[15] M. Klein, "iDCSS: Integrating Workflow, Conflict and Rationale-based Concurrent Engineering Coordination Technologies", CERAs, 3, 1, 1995.

[16] Y. Miao, J. Haake, Supporting Concurrent Design by Integrating Information Sharing and Activity Synchronization",

[17] J. Lonchamp, B. Denis, "Fine-grained process modelling for collaborative work support: experiences with CPCE", Journal of Decision Systems, 7, Hermès, pp.263-282, 1998.